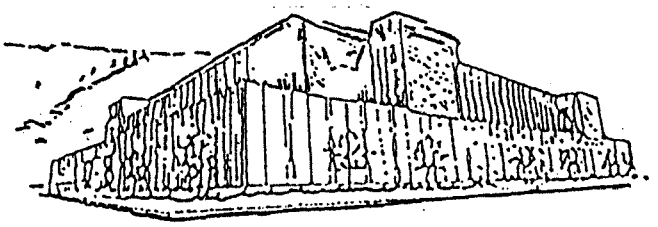
1998

Process in user interface development

Srinivas V. Mondava
The University of Montana



Maureen and Mike
MANSFIELD LIBRARY

The University of MONTANA

** *Please check "Yes" or "No" and provide signature* **

Yes, I grant permission X
No, I do not grant permission ____

Author's Signature M Simmons

Date 06/22/98

A PROCESS IN USER INTERFACE DEVELOPMENT

by

Srinivas V. Mondava

B.S. Shivaji University, 1988

Presented in partial fulfillment of the requirements

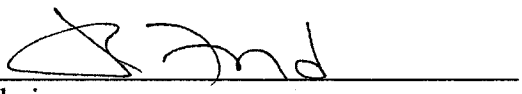
for the degree of

Master of Science

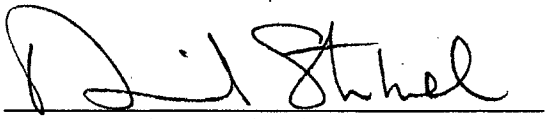
University of Montana

1998

Approved by:



Chairperson



Dean, Graduate School

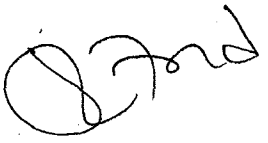
6-23-98

Date

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

Mondava, Srinivas V., M.S., June 1998 Computer Science

A Process in User Interface Development (52 pp.)

Director: Raymond Ford 

Most of today's applications are based on graphics-based systems with inherent complexity in design and development.

This project presents a process in User Interface development using spiral model and object-oriented principles in design and development. The purpose is to put well-defined software engineering principles into practice.

CONTENTS

CHAPTER 1

OVERVIEW

1.1 Premise

Object-Oriented Methodology and Graphical User Interfaces have become a fact of life in the software industry today. The power of the visual paradigm is now widely recognized and is the foundation for graphics-based systems. However, rising expectations of usability and the increasing complexity of software make the design, development, and support of user interface based software difficult. The production of high quality user interfaces requires both an iterative development process and the use of a formal development and evaluation methodology. Various user interface evaluation techniques have been proposed to optimize user interface development, but they don't seem to fit well when there are other system components that need to be integrated with the user interface. This can be disconcerting to software developers, especially when it is obvious that there are risks involved in user interface development.

This project illustrates how a user interface development and evaluation methodology can be adapted to fit as an integral part of Bohem's spiral model of software development [1], by associating Bohem's concept of risk with the costs of user interface development. The project in focus here is the development of a graphical user interface for the Ecosystem Information System (EIS), a system designed to provide access to a network accessible repository of information of interest to ecosystem modelers. Although it is always better to have a principled approach to development, the question is, what

principles? Gould, Boies, and Lewis [2] suggested following general principles in user interface development.

1. Development should include early and continuous empirical testing, centered around appropriate users performing representative tasks.
2. As the development proceeds, it should incorporate subsequent *iterative refinement* procedures and *cost/benefit analysis* to determine the most cost effective changes to make to the user interface design.

The concept of early user testing revolves around rapid prototyping methods. This gives the users and developers a chance to visualize a complex concept like EIS for its purpose and use. Once a prototype is developed, evaluation of its operations is done by the user, while the developer looks into the design and development methods used in the construction of the prototype. One method for developing a prototype is the use of an interface design toolkit. This approach gives quite a bit of importance to rapid prototyping at early stages of user interface design and development because of the relative ease with which interfaces can be constructed. It is valued from a research-oriented perspective, because the toolkit becomes a valuable exploratory tool. It also addresses the concept of examining the composition of the toolkit itself and evaluating its effectiveness.

The following chapters will look into the development process and prototype evaluation with respect to operations and the development tools.

CHAPTER 2

GENERAL THEORY IN A UI DEVELOPMENT

Before developing a UI for a data repository that is organized in an object-oriented manner, it is important to understand the software development process model that will be used to guide the complete system development. Development methods for developing a UI will also be explained in this chapter.

2.1 Using a spiral model of software development

A process model allows the persons responsible for the development of the software product to identify and order various stages of development, and also to identify transition criteria from stage to stage. There are various process models that can be used to design different stages of a software development product. The criteria to select one of the process models involves understanding the strengths and weaknesses of each process model and how well it can be adapted to further the goals of product development.

The process model used here is Bohem's spiral model of software development [1]. The concept behind the spiral model is that there are a number of cycles of development. Each cycle involves a progression that addresses the same sequence of steps. Bohem's model suggests that the developer first identify elements of the system whose complexity or uncertain requirements pose the greatest risk to its development. The developer should then focus the most attention in early cycles to reducing the risk related to these components. A typical 2-D spiral model design begins with the

identification of (1) the objectives or goals of a portion of the product (e.g. functionality, performance, tools); (2) alternate ways to achieve the implementation of this product (e.g. design A vs. design B); and (3) constraints imposed on the application of the alternate ways of implementing this component of the product (e.g. costs, labor, time). The next step is to evaluate the alternate ways of achieving this goal, in the context of the objectives and constraints. This step helps the developer to identify risks associated with this portion of the product. Next the developer identifies ways to resolve risks, such as by using rapid prototyping, formal analysis or user questionnaires. The developer then balances the risks against the cost of resolving risk. Focusing on resolving the riskiest elements first helps in planning multiple development cycles or iterations in an orderly and cost effective way.

In a visual depiction of a spiral model, the process stages are depicted as quadrants. Axes are quadrant boundaries. The development path is usually linear, but can discontinuous or non-sequential. The interpretation is that the activity depicted in quadrant Q1 is completed first, then Q2, etc. Eventually the Q1 activity is repeated, but now in the context of the structure designed or information gained in the first cycle.

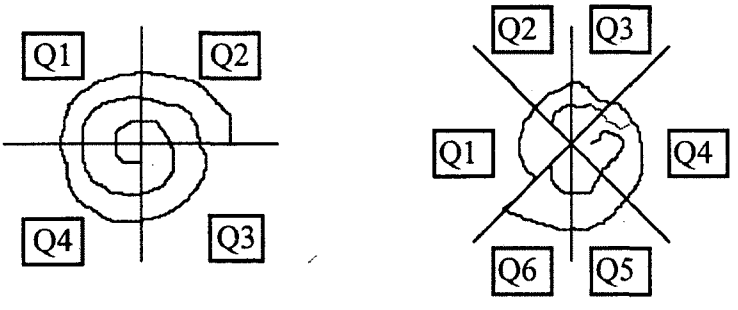


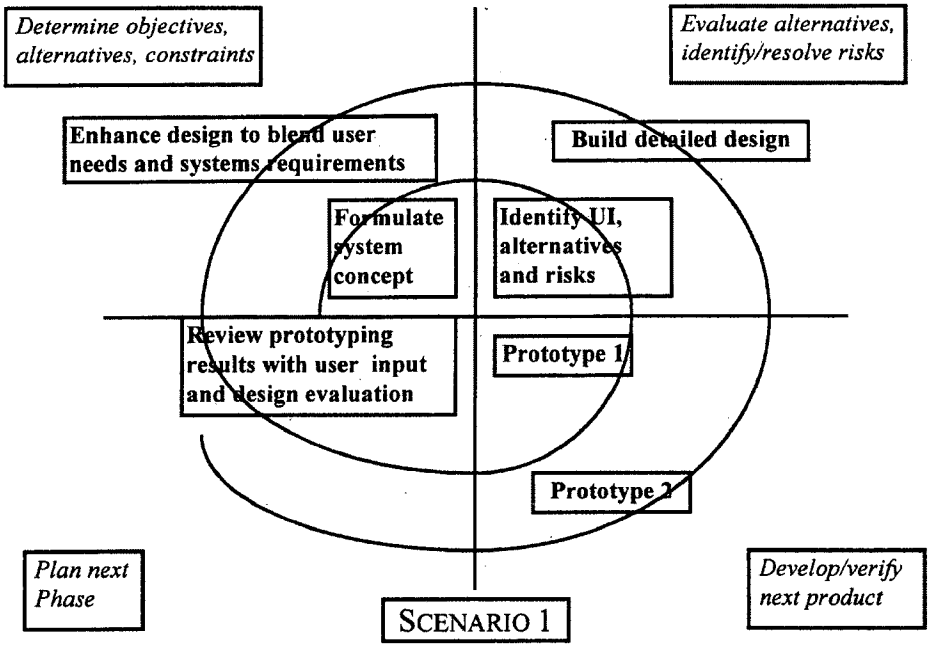
Figure 1

One scenario for the development of a UI requires that specifications be gathered by understanding the goals of the system, determining who the users are, and addressing the specific tasks the users expect to accomplish using this system. Eventually, the developer must design the product, balancing user needs with system needs, and assess the usability of the product by testing a prototype with the users. The development and refinement processes are repeated until the optimal blend of user satisfaction and overall system functionality is achieved. This approach emphasizes early user input on the aspects of usability, with user feedback documented and the interface modified to meet the needs.

Another scenario for the development of a UI might require the developer to do independent prototyping efforts using different development methods. For example, one prototyping method is to use a UI generator to automatically convert visual layout of displays to interface code. The visual layouts of the interface can be created interactively by positioning and setting display objects as supported by the UI development tool. In contrast, another prototyping method is to use a programming language and windowing library to develop the user interface. These two methods are explained in Section 2.3. Each of these scenarios can be considered as a form of spiral development. Both involve the use of "best guesses" followed by weighing the constraints over the objectives, designing a prototype and evaluating the result. The functional objectives of the system can be analyzed using the prototype allowing the goals to be refined. The process can be repeated again using the information gained.

For example, in Scenario One understanding all user needs in the first cycle of the spiral is impossible. The developer must make some judgments, get user input, and base a preliminary set of goals on this partial information as to what the overall system can achieve. From this a prototype can be designed. The prototype can be used to develop a better understanding of user input, user needs, and the user and developer roles in defining the possible structure of the system as a whole. Based on this improved understanding, new goals can be set and reviewed, leading to the next cycle of development.

Unfortunately, developing a UI using scenarios like the one mentioned above is not always so systematic, because of constraints in costs, time and technological limitations. However an approach like the Bohem's spiral model emphasizes how the development process weighs such constraints against its objectives. For the development of a UI with its inherent complexities, the risk-driven approach of the spiral model balances the risks associated with the UI product. The risks associated with UI development can span from clarifying vague functional goals to helping to stabilize evolving user expectations, in terms of performance, flexibility and security concerns. A risk-driven strategy helps to point out errors and unattractive alternatives early, so that the developer can concentrate on better approaches to the development of this product, rather than making premature arbitrary decisions. It also helps the developer to plan the amount of time and effort that should be devoted to various aspects of the product development, like quality assurance, formal verification and testing.


Determine objectives, alternatives, constraints
Evaluate alternatives, identify/resolve risks
Enhance design to blend user needs and systems requirements
Build detailed design
Formulate system concept
Identify UI, alternatives and risks
Review prototyping results with user input and design evaluation
Prototype 1
Prototype 2
Plan next Phase
Develop/verify next product
SCENARIO 1

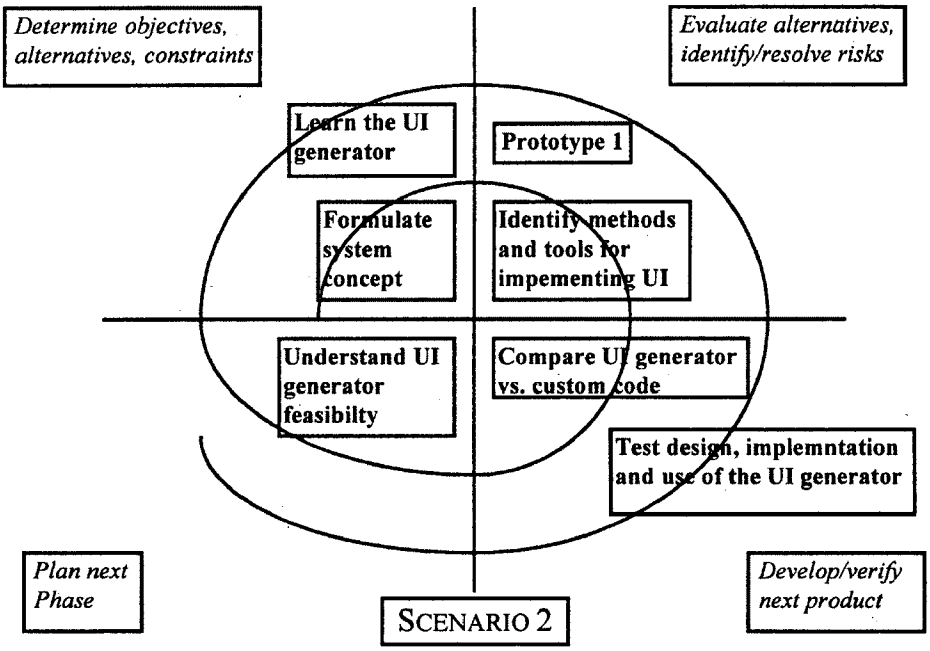

Determine objectives, alternatives, constraints
Evaluate alternatives, identify/resolve risks
Learn the UI generator
Prototype 1
Formulate system concept
Identify methods and tools for impementing UI
Understand UI generator feasibilty
Compare UI generator vs. custom code
Test design, implemntation and use of the UI generator
Plan next Phase
Develop/verify next product
SCENARIO 2

Figure 2

2.2 Object-oriented design approach

According to Booch [2], an object-oriented design encompasses the process of decomposing a system into objects, using a notation to depict both logical (classes and objects) and physical (module and process architecture) structures, and developing static and dynamic models of the system. An object-oriented design method emphasizes the proper and effective structuring of a complex system.

An object is a tangible entity that exhibits some well defined behavior, whereas a class is a set of objects that share a common structure and a common behavior. There are four major aspects of an Object Model.

(1) Abstraction denotes the essential characteristics of an object that distinguish it from other kind of objects.

(2) Encapsulation is the process of collecting all the details of an object and hiding all its non-essential details. Each object in effect should have an interface part and an implementation part. The interface part focuses on the outside view of the object, i.e., the object's essential behavior, but the implementation part is encapsulated and hidden from the users of the object.

(3) Modularity is a process of decomposing a system descriptions into a set of "module" descriptions. The idea is that each module is simple and can be understood fully so that change within a module can be made without affecting other modules.

(4) Hierarchy is a process of ordering abstractions in a tree like fashion. Inheritance is the most important aspect of object-oriented hierarchies. Inheritance implies that

each class/object shares the structure or behavior defined for each of its ancestor classes/objects.

The UI being developed here is for a system that is to be used to define a distributed, object-oriented data repository. Because the object-oriented paradigm is the basic organizational tool for the data repository, the software and its UI should support accessing, building and querying the data repository in object-oriented terms. Suppose the data collection is organized by a hierarchical structure known as class hierarchy where each point in the hierarchy is a class definition which represents a meta-description of a particular dataset. Each meta-description includes the description of data attributes, the description of operational components that are used to access the data, and other information about the data. Typically, basic operations are required for assessing, modifying and adding to nodes in a class hierarchy. A good UI must support operations that will encourage use by both a skilled object-oriented designer and a novice.

2.3 User interface development

The development of a UI consists of two parts: the user interaction component and the base software interface. The user interaction component defines how a user interface works, i.e. the "look and feel" that provides what a user sees and hears and the "behavior" that describes the effects of the user's actions. The base software interface is the means through which the UI actions are actually linked to the code that implements the functions whose behaviors are observed by the user.

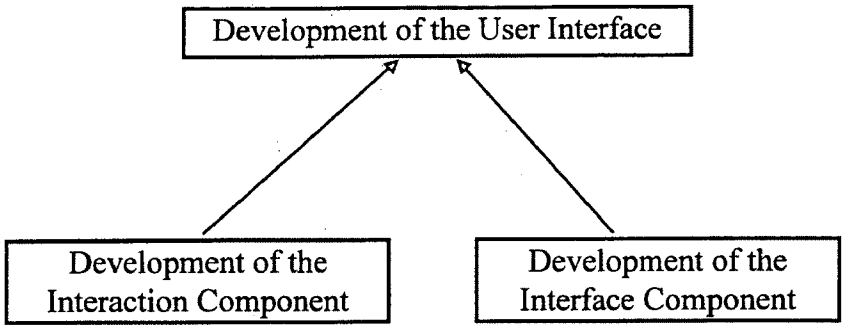


Figure 3

In essence, the interaction component (or the behavioral domain of a user interface) looks into human factors like guidelines and rules, cognitive limitations and interaction styles. This provides a way of enhancing human use of an interface. The base software interface (or the construction domain) involves widgets, algorithms procedure libraries, control and data flow, event handlers, callbacks, object-oriented design principles and use of UI languages.

A good UI development must address both behavioral and implementation issues as shown in Figure 3. This section focuses on the implementation component of an interface development. Tools involved in the construction process include machines, computer science principles, and software engineering principles. The major players are software designers/ researchers, software engineers, programmers and, if I may, graduate students or others assigned to implement designs.

The UI that is elaborated here is called the Graphical User Interface (GUI). The general purpose of a GUI is to provide a window-based interface for user selection of alternatives, other user inputs, and display of output. A program using a GUI has two

parts, the visual part (front end) and the application code. Dividing the system in this manner allows independent development of each part. When designed properly, the application code can be used with different front-end interfaces. The GUI itself is comprised of a visual part embodied in the display of static screens, and events (implemented via callbacks or/and eventhandlers), which are system actions associated with certain user actions. A GUI provides the system with an event-driven approach. That is, the system processes events as they happen, and implements corresponding actions. Some events result from user actions (keyboard press, mouse click, etc.) whose order is unpredictable. Thus a GUI must support asynchronous handling of user actions, emphasizing the availability of various actions at any given time. The sequencing within each action is independent of sequencing with other actions. For the development of a base GUI, one approach involves working in an X Window environment, writing code using toolkits and a callable library of routines. Another approach involves the use of so-called User Interface Management Systems to generate the GUI.

Developing a GUI in an X Window environment requires the developer to manage both the look and the actions of a GUI through a library of callable routines that implements visual features. Several X Toolkits are available to provide code for various interaction techniques by using different physical devices entering specific types of input. A widget is a basic data type or object for a physical device that encapsulates the "look and feel" for a user to interact. A widget class holds information on graphical objects. Instances of these widget classes help build the visual part of a GUI. Examples of widgets are text widget, buttons, menus, etc.

Toolkits interact with application code using callbacks. The actions associated with each GUI event are assumed to be collected into a set of procedures, and callbacks represent the invocation of an appropriate procedure by the event handler in response to the "next" action taken by the user. That is, when a user interacts with a widget, the interface code links to the correct application code by an event, which prompts a callback. Through the proper management of events and callbacks, the user can interact with several tasks or applications, each in a different window.

One of the most popular windowing systems on Unix systems is the X Window System. An X Window System provides a toolkit, called Xt and a library, called Xlib, for interface development. The Xlib library has low level callable procedures. Xt is at a higher level, consisting of procedures built on combinations of Xlib procedures. Xt defines a very basic set of widgets (most of which are functional, and not visual), providing a foundation to build the real widgets that are used to construct a GUI. The style and procedures to build even higher level widgets with both functional and visual components are called interaction style support tools. These tools enforce a particular consistent interaction "look and feel" to a user interface. Two of the most popular commercial interaction style support tools are OSF/Motif and OpenLook.

Combinations of all these toolkits, libraries of routines, and interaction styles can be used to develop a GUI. This method of user interface development allows rapid prototyping by connecting ready-made interface components. However, no matter what level tool is used, programs are developed as code containing complex sequences of the procedure calls that are used to access and incorporate Xlib, Xt and Motif facilities.

A User Interface Management System (UIMS) is designed to provide a non-programming approach to GUI development. A UIMS tool is an interactive system that supports the various processes involved in developing interfaces, including design, prototyping, execution, evaluation and maintenance of the interface. A typical UIMS is built on a particular window toolkit and supports a particular interaction style. The fundamental purposes of UIMS are to allow the system to be developed with a minimum of writing code and to support all activities in the interface development cycle. For example, the UIMS TeleUSE is built on X Window system, comes with an Xt toolkit, and supports various widget sets like OSF/Motif, MIT/Athena and others. Some of the important characteristics of a good UIMS is described here. A good UIMS should provide a non-programming environment that helps in layout of displays (screens), including defining, positioning, resizing and setting of the display, object attributes, and defaults. It should also provide support for linking the visual elements and their objects in the application code via something other than traditional programming. For example, direct manipulation in TeleUSE is done using D scripts. These scripts are written using a compact scripting language called D. The purpose of these scripts is to "manage the dialog" between the visual elements and the application code. A good UIMS should support runtime mechanisms for sensing user actions on objects in a display and provide feedback in a display. It should also have a generator for producing interface code from interface definitions as well as ways of storing and managing the generated code.

In the following chapters, these two approaches to GUI development are further explained in the context of their use in the development of a real GUI.

CHAPTER 3

THE TARGET APPLICATION SYSTEM

This chapter looks into the operations and functions of the GUI for the Ecosystem Information System (EIS). The goal of EIS is to provide a data repository of ecosystem modeling components, organized in an object-oriented paradigm. The GUI front end for EIS must therefore support assessing, building, and querying the data repository. By defining general operations to meet the functionality of the GUI, specific functions can be designed and implemented. One such implementation of the GUI is discussed here.

3.1 EIS and the role of GUI

Modern ecosystem management and analysis demands information sharing between a number of organizations. Present computer and networking technology makes this sharing possible, but the information organization to make sharing practical is still lacking. As described in [3], the goal of EIS is to create a network assessable repository of ecosystem information that provides access to datasets and models in a user-friendly fashion, allows the distribution of locally created material and contribution of materials from outside users, and is populated with material sufficient to illustrate both its use and its value to potential users.

Free distribution and access to information is paramount to EIS, because ecosystem modeling and management involves collaborations between individuals housed in different departments, working for different organizations located in different locations, yet demands rapid inter-change of information ranging from simple messages

to parts of complex programs to large datasets. Thus, EIS must provide a set of services associated with the management of a distributed and object-oriented data repository. Operations to create, access, and share the distributed data repository must be designed according to an object-oriented paradigm. As described in [3], EIS is designed based on four major object-based subsystems: the OME (Object Management Engine), GUI (Graphical User Interface), ORB (Object Request Broker) and ODBM (Object Database Manager). Our primary focus here is the GUI.

The role of the EIS GUI is to provide the EIS user with the ability to create class definitions, define class attributes, place a class in a hierarchy of other classes, and create instances of a class. Each such operation in the GUI must support EIS user interaction in appropriate dialogs. The GUI should present the set of supported operations to the user, detect and interpret user selection, solicit the designation of the appropriate operation parameters, perform simple "screening" on operation parameters, and resolve operation requests into specific EIS function requests that are passed to other EIS subsystems. That is, because of the distributed nature of EIS, GUI itself does not invoke methods, but passes the screened function requests to other EIS subsystems for execution. It then waits for operation results to be returned, and presents the results to the user.

Hence the basic object manipulation operations that should be presented to the user include the following: select and browse a class hierarchy, register a class/subclass definition, register a method definition, register an instance definition, record and

possibly "log" user selection, solicit designation of appropriate parameters, and perform simple screening on operation selections and parameter specifications.

3.2. Design of GUI

After the basic operational criteria for the EIS GUI are identified, specific functions can be developed to meet these requirements. This process of subdividing a complex subsystem like the EIS GUI into operations and functions helps in implementing a prototype. Using a prototype the developer can analyze the GUI subsystem with respect to how it meets its functional objectives and how it must interact with other subsystems, allowing modifications if necessary. This section includes two tables. Table 1 identifies individual user level operations that together support the functional requirements of the EIS GUI. Each of these operations is subsequently defined in terms of sequences of internal functions. These functions help realize the requirements for building a prototype of the EIS GUI. Table 1 provides a cross-reference between the GUI operations and the set of internal functions that support the GUI operations. For example, a GUI operation "Hierarchy Browsing," can be supported using the following functions: get_context, get_hierarchy_list, list_hierarchies, scan_the_list and select_a_hierarchy. Functions that are identified here are broad-based. All of these functions may not be implemented because some of them are inherent to the tools that are used to implement this prototype. For example a function like "scan_the_list" is assumed to be provided in the form of "scroll bars" for window based tools that allows a user to interactively scan a list.

The second table, Table 2, uses the broad-based generic functions of Table 1 to define specific functions that together will help implement a prototype for GUI. Each of these functions has a specific name, a list of arguments it requires, and a description of the results it will produce. This table gives a more specific implementation perspective for the development of a GUI prototype.

With the help of these tables, we achieve a structured way of breaking the subsystem GUI into specific components. Looking into the Bohem spiral model, these tables represent first quadrant activities focussed on identifying and defining the goals for the first spiral of GUI design and prototyping. Instead of describing each individual operation and function that makes up the GUI, these tables provide a self-descriptive set of operation and function names, and a cross reference between each operation and function and between a function and its attributes. For example, one of the primary requirements of GUI is to provide a way for the user of EIS to create a class and assign this class in a particular hierarchy. This hierarchy can have local, temporary, or global contexts. To meet this particular EIS GUI objective, operations to implement parts of the "Hierarchy Browsing" and "Create class/subclass" as internal operations are defined and cross referenced across two tables. Thus, an example sequence of functions that can be used to implement class creation could be: get_context, get_hierarchy_list, list_hierarchies, select_a_hierarchy, select_object, create_object, print_hierarchy.

Table 1

Operations	**Functions**
Hierarchy Browsing	get_context
	get_hierarchy_list
	list_hierarchies
	scan_the_list
	select_a_hierarchy
Create class/subclass or	get_context
Create method or	get_hierarchy_list
Create instance or	list_hierarchies
	scan_the_list
	select_a_hierarchy /open
	select_object
	create_object
	print_hierarchy
Editing operations or	select_object
	cut_object
	copy_object
	undo
	paste_object
	delete_object
Print Operations	print_obj
	print_hierarchy
	print_tree
Register operations	save
	save_as
User input operations	input_values_for_object

Table 2

Functions	**Arguments**	**Return Values**
OPEN	hierarchy_id	hierarchy
SAVE	hierarchy_id	null
SAVE_AS	hierarchy_id, new_name	null
CREATE_OBJ	hierarchy_id, object_description	object_id
DELETE_OBJ	hierarchy_id, object_id	hierarchy
CUT_OBJ	hierarchy_id, object_id clipboard_address	hierarchy
PASTE_OBJ	hierarchy_id, object_id clipboard_address	hierarchy
COPY_OBJ	hierarchy_id, object_id clipboard_address	null
SELECT_OBJ	hierarchy_id, object_id	object_description
OPEN_OBJECT	hierarchy_id, object_id	object_description
PRINT_OBJ	object_id	object_description
PRINT_HIERARCHY	hierarchy_id	hierarchy
PRINT_TREE	hierarchy_id	tree
UNDO	clipboard_address	null
GET_HIERARCHY_LIST	null	list_of_hierarchies
SELECT_A_HIERARCHY	hierarchy_id	hierarchy

3.3 First spiral in GUI development

Vague functional objectives and changing user needs and expectations make developing a user interface for an EIS system an inherently complex process, which demands a good UI design methodology. Having a framework that focuses specifically on the UI aspects of requirement analysis, testing, and evaluation could make this task easier. Creating an effective GUI for a complex application system like the EIS requires an iterative process of identifying operations, formal analysis of these operations, software prototyping, and review. This approach plans on multiple development iterations as is also evident in Bohem's spiral model. Initially, we take the best known operations that can be integrated into the development of GUI and implement a prototype. Using the Bohem's spiral model first quadrant involves designing a prototype. Tables 1 and 2 of Section 3.2 help with quadrant 1 activities related to identifying the set of object manipulation operations that can be used to develop a prototype of EIS GUI. Second quadrant activities involve using this information to develop a prototype using some specific development tools. Subsequently, quadrant three and four activities involve evaluating the prototype with respect to EIS functional objectives and other criteria, then refining the objectives and planning further spirals of development.

The first prototype of the EIS GUI was developed using a very informal version of the information in Tables 1 and 2. The purpose of this prototype was to look at operations required to create a class with values, such as name, attributes and operations, and place this class within a hierarchy of other classes. In this first prototype, interactions

with other EIS subsystems were more or less ignored. This prototype was implemented using a User Interface Management System tool called the TeleUSE GUI generator. This generator divides the visual (screens) part of the UI from the application code to allow the GUI prototype developer to rapidly construct the visual display without details of the dynamic part. The visual part is developed using the TeleUSE Visual Interaction Programming layout editor (also called VIP). VIP allows a developer to choose different Motif/TeleUSE window objects (widgets) and helps arrange these objects interactively. Once the screens or visual layouts are arranged according to the requirements, the layout can be converted to interface code. Subsequently, another component in TeleUSE can be used to define the interaction between the visual layout objects and the application code. This so-called dynamic part is handled in TeleUSE using a Dialog Scripting Language. By combining the visual component, the dynamic component, and the application code, an executable system can be generated using user interface builders. This method of development helps in rapidly generating the prototype for further analysis and review. However, the first prototype focused exclusively on the visual component, leaving the dynamic component and application code to interact with other objects for subsequent efforts.

The first prototype focused on instantiating a GUI on a Unix workstation. The visual form was based on three paned windows as shown in Figure 4. The first has a pull down menu with options like FILE, EDIT, VIEW, OPERATIONS, and HELP menus. FILE menu items included SAVE, PRINT and EXIT options. OPERATIONS menu items

include CREATEOBJECT, OPENOBJECT, and DELETEOBJECT options with dialogs to solicit object information such as name, id, parameters, attributes and operations. VIEW menu items include LOCAL CONTEXT and GLOBAL CONTEXT. EDIT menu items include UNDO, CUT, COPY, and PASTE options. Below this main menu is a main window, used to display the current object hierarchy tree structure.

The second paned window was envisioned as another way of implementing an object hierarchy tree, using pushbutton widgets to represent classes. However, this was never completed.

The third paned window contains a scrolled text field, used to display the textual form of the "current" object. In an object hierarchy the current node is highlighted with a black box around the name. The textual description of this current object is maintained in the third window to provide a context in which editing operations like DELETE, CUT, COPY, PASTE can be performed. Each object hierarchy has a root node that cannot be deleted.

Using Table 1 and 2 as a basis for operational criteria of the EIS GUI, this first prototype provides a very basic subset to meet the functional requirements. It tries to cover a broad category of operations, but only a subset of functions. All the operations mentioned in Table 1 are taken into consideration while designing this prototype. However functions that are provided achieve very rudimentary aspects of these operations. Prototype 1 achieves the goal of creating a "shell" of the GUI for EIS system but leaves important questions, such as interaction with other subsystems or objects,

application code to implement the required functionality, etc. unanswered. This is the starting point for further spirals of development.

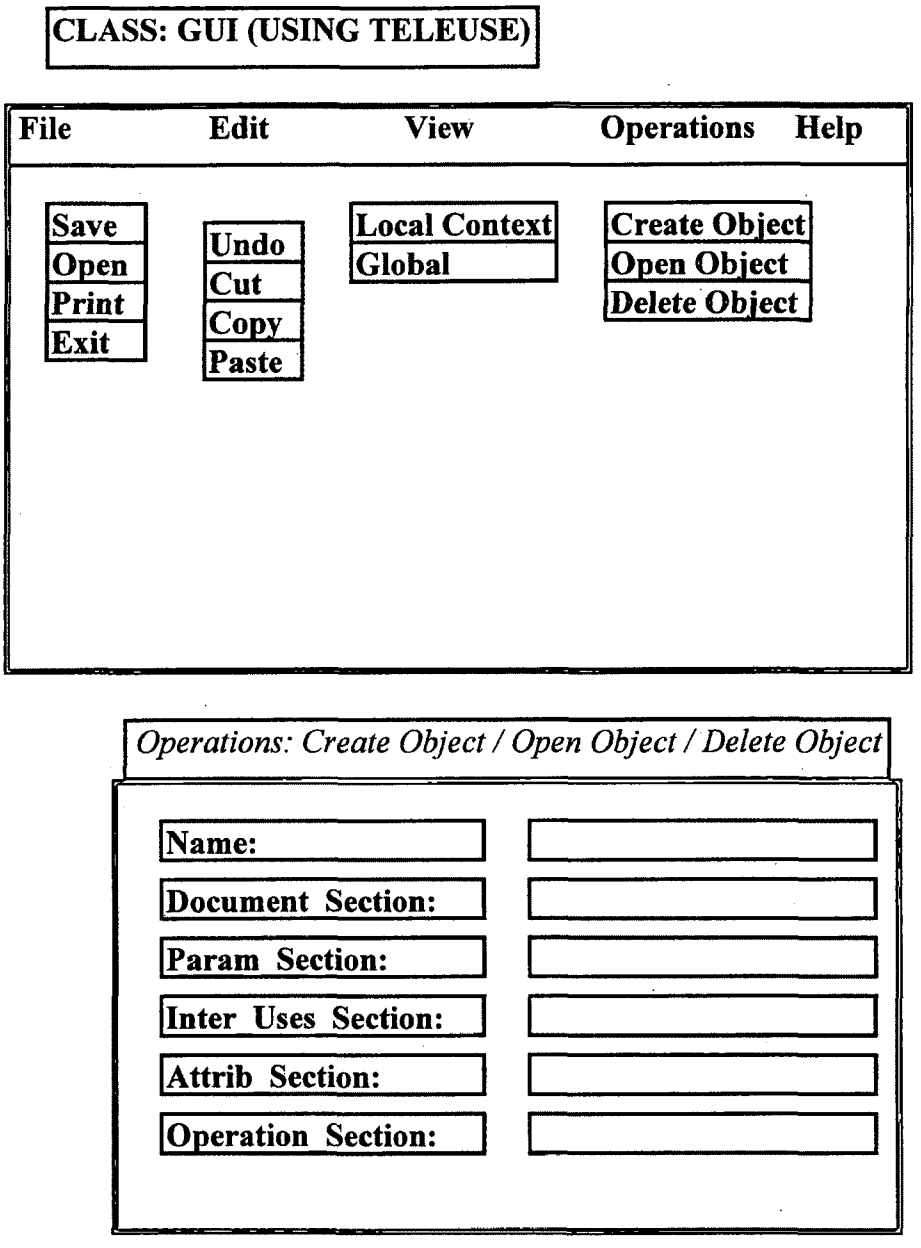


Figure 4

CHAPTER 4

EVALUATION AND FURTHER DEVELOPMENT

With the creation of Prototype 1 for EIS, system development enters the fourth quadrant, where activity focuses on evaluation of the functionality and the tools used to develop this prototype. This chapter summarizes fourth quadrant activity based on the analysis of prototype 1 with respect to usability, preliminary specifications for the target EIS system along with operational requirements and the development tools used. Based on the evaluation we propose changes in the system form, function and GUI operational criteria. We also describe other tools that can be used in the next development process.

4.1 Evaluation of operations in Prototype 1

Before we go further into the operational evaluation of Prototype 1, due consideration needs to be given to the inevitable process of GUI development, where problems with certain operations often are detected much later in system operation than would ordinarily be the case. The idea is to develop and use relatively standard GUI tools to prototype what are perceived as relatively simple GUI elements, and then evaluate the GUI in the context of evolving information on other aspects of system functionality. This evaluation for the GUI exposes problems stemming from missing functionality, confusing dialog flows, and other problems. The goal is to assess, at this point, what the user needs and expectations are, and try to translate them into operations that are defined for the next stage in the development of GUI. In this case the primary users and evaluators were a software design class of students. The possible GUI problems and system errors

uncovered during interface evaluation are characterized as follows.

1. missing functionality and dialogs.
2. ineffective cross-checking or screening
3. inadequate or no help information provided.
4. lack of configuration capability.

User operations provided by this prototype are very rudimentary in nature, with lots of limiting assumptions. For example, in an operation like "Create class/subclass", the template to create a class allows very basic attributes like name, etc., and the parent is assumed to be the "current" selected class in a hierarchy tree. Operations like "Browse Hierarchy" are not included. The various "Tree Operations" have limited functionality and do not provide necessary checks. For example, the function "delete_object" deletes an entire subtree, and does not provide any "confirmation" dialog. Also, many operations are not well phrased with respect to object-oriented concepts such as classes, instances, and methods, and fail to guarantee consistent interrelationships and dependencies among these objects. Finally, there is no checking for consistent identifier definitions and uses, which makes the overall usability of the system quite difficult to assess.

Following are some additional comments from the initial evaluators on the current functionality, with respect to operations or modifications needed. The first name indicates the menu in which the operation appears, followed by the name of the operation itself.

1. File - Save: Should be implemented, with hierarchy saved in an appropriate format.

2. File - SaveAs: Should have an option to save a hierarchy from current format to any other appropriate format.
3. File - Print: Should distinguish between typing on paper vs. "printing" in one of the panes of the GUI window.
4. File - Close: Should cause GUI to close current file. If any change is made it should allow saving the modified file.
5. File - Get-heir-list: Should get list of all hierarchies available, allow user to select a hierarchy, then open it.
6. File - New-hierarchy: Should allow the ability to start a new hierarchy.
7. File - Get-host-list: Should get a list of all hosts available, allow user to select a hostname, then connect to that host.
8. File - Open: On selecting this option, the GUI should ask the user to specify the file name.
9. File - Import: Should allow the user to save in local host.
10. File - Export: Should allow the user to save in global host.
11. File - Exit: Should have an option of saving before exiting.
12. Edit - Undo: Could have multiple levels of undo.
13. ObjectMenu - Create Class: As of now it accepts a node without a name.
14. ObjectMenu - Create Instance: This is necessary to create an instance of a class.
15. ObjectMenu - Create Method: This is necessary to create a new method.

16. View: It should have more than two contexts, such as workstation or lab contexts, to mention a few.
17. Help: Should provide documentation on the menu items.
18. Clear: Need this operation for clearing the 'undo' buffer area or the tree on the screen
19. Execute: Need to execute an instance of a class.

Some additional comments are listed, regarding functionality that needs to be added, cosmetic aspects of the displays, and other characteristics.

1. Need a color file that GUI uses for its widget colors.
2. Provide simple screening of new class information.
3. Have a BROWSE operation.
4. Limit global data editing.
5. User should have access to the clipboard.
6. Duplication of the node names should be avoided. Copy and paste allows this.
7. Object information dialog box should allow the user to input all the fields without using the mouse.
8. Move "Delete Object" from Operations Menu to Edit menu.
9. Add Encapsulation operation on the Tree, for example, Hide_Method, Hide_Instance, Minimize_Subtree, etc.

4.2 Evaluation of tools in Prototype 1

Prototype 1 is developed using a GUI generator, TeleUSE UIMS. To understand the methods and processes that are used to develop this prototype using the GUI generator, an additional mini-spiral is necessary for learning and understanding this tool. This process allows a developer to look at the tool primarily with respect to its usability and functionality, considering the fact that UI design and development typically mirrors the stages of a system's entire life-cycle: requirement analysis, design, evaluation and iteration of versions or prototypes until the final product is delivered. It is important to analyze the requirements that a GUI generator should have, for reasons of evaluating the ability of the present generator (TeleUSE) to meet this project's long-term development goals. That is, the purpose of the requirements analysis of the GUI generator is to provide information to allow a decision to be made to continue further development using this particular GUI generator, or to switch to producing GUI-version 2 with direct X/Motif coding.

Following are some of the important characteristics that are considered in analyzing an Interface Development tool.

1. Functionality of the tool.

   The functional capabilities of a tool are the most important characteristics. It should have the ability to produce the visual displays and the interaction styles required or expected by users of an interface.

2. Usability of the tool.

An interface tool building system should have an easy to use user interface itself. There is a trend with user interfaces to have functionality become inversely proportional to the usability of the tool, which should be avoided at all costs. A tool with more functional systems need not be harder to use. Also a GUI for the tool needs to weigh against the amount of programming that is done in order to construct an interface. This should take into account the broader variety of interaction styles that is supported without a programmer writing code.

3. Ability to produce direct manipulation interfaces via direct manipulation.

The tool should be able to produce direct manipulation interface features via direct manipulation in the tool interface. A tool that forces the interface implementor to resort to programming for different kinds of interfaces may not be the right tool for this type of development environment.

4. Styles supported by the tool and customization of style.

This is closely related to the functionality of the tool. Even if a tool comes with all the widgets sets and interaction styles that are available, there could be a need to create a special widget for the purpose of this software product. Thus, it is necessary that the widgets and the widget set be customizable.

5. Creation of dynamic interaction objects.

Development of dynamic (runtime changeable, data dependent) interaction objects is an important requirement for all but a very simple GUI. An interactive

development tool should support dynamic objects.

6. Support for evaluation and iterative refinement.

A tool should provide direct support for evaluation and iterative refinement in a more direct way than simply claim that modifications to an interface are easy to make using the tool. For example, a tool might allow internal implementation of an interface, then support collection and analysis of quantitative and qualitative user feedback about that prototype.

7. Type of control structure and callbacks.

The type of control structure imposed by a tool on software is important when the resulting GUI is connected to existing software or to software being developed by other groups. For example, the object oriented EIS design illustrates such a scenario. The ability to define and manage callbacks is also an important part for coupling with other software. A tool can embed its output into existing software or embed existing software into the output of the tool. It can also generate code that maintains control of an event loop and decides when and how to respond to user actions.

8. Portability of the interface produced by the tool.

Often it is desirable that the interface developed by a tool be able to run on different platforms. A tool providing portability by keeping the same look and feel across different platforms is not very desirable. Being tightly integrated with

a standard windowing system (i.e., X, Windows, etc.) tends to make a tool less portable.

9. Changes to GUI design independent of code generation.

A tool should allow the interface developer to make changes to already generated code for the user interface, without generating completely new code. Also the process of achieving such changes should be easy to accomplish.

10. Runtime performance of the interface produced by the tool.

The output produced by the tool can be compiled or interpreted. A compiled interface is almost always faster than interpreted code. This defines the concept of runtime speed, which in turn affects the usability of an interface.

11. Cost, documentation, and customer support.

Cost of these tools can determine whether it is feasible to use for the development of a software product. The quality of documentation helps developers understand the tool. Customer support helps to enhance the usability of the tool.

Before a new prototype is developed, it is necessary to look at TeleUSE UIMS and see what development aspects (good or bad) are part of this GUI generator. This helps a developer make an intelligent choice as to whether to continue to use the development tool. For example, a GUI builder like TeleUSE imposes limitations on which platforms and what windowing system the code it generates can use. In this evaluation of the tool, emphasis is also given to the ease or difficulty with which this tool can be used to develop the target software product. This point is important because of the

fact that revisions or iterations to GUI prototyping are done possibly by different individuals or groups. That is, every time a new developer or programmer becomes responsible for extending the code, additional time and effort needs to be spent in understanding this prototyping tool.

Properties of TeleUSE, its advantages and disadvantages.

TeleUSE belong to a class of software called User Interface Management Systems (UIMS). It is workstation-based and requires the X Window System and a window manager. The product provides a collection of tools and libraries to help develop the display (static) and interactive (dynamic) portions of GUI. An application program with a GUI has two parts, the user interface itself and the application code. As for the user interface code, it too has two parts, static part and a dynamic part. The static part is called the presentation component. It contains the widget hierarchies used for a specific application's user interface. It also describes the screen layout of the user interface. This development in TeleUSE is done using a Visual Interaction Programming layout editor also known as VIP. This lets the developer of a user interface choose different widgets and arrange them according to the application requirements. The VIP layout editor is a so called WYSIWYG editor. Files can be saved in native TeleUSE format or in 'C' or 'UIL' code. The dynamic part of the user interface, also called the dialog manager handles the interaction component between the presentation component and the application code. TeleUSE provides a dialog scripting language called 'D Scripts' to help define this interaction. These files are saved in a native TeleUSE definition file. Once the static part,

the dynamic part and application code are done, an executable is generated using the User Interface Builder.

TeleUSE UIMS does a good job in isolating user interface code from the application code, thus making the logic of each part easier to understand, develop and maintain. It supports various toolkits and allows customization of widgets or interaction styles. The dialog management part ('D' scripts) is compact, so that developing the presentation component is very handy if the whole interface and event handling part is done in TeleUSE. Converters are provided to allow both the static and dynamic parts of the user interface to be converted to 'UIL' or 'C' code. TeleUSE in essence provides a tool that lets the developer do rapid prototyping.

However this concept of isolating the user interface code with application code is not new. Following is a brief outline of the drawbacks in using TeleUSE.

1. TeleUSE runtime libraries provide a set of functions that is supposed to make the programmer write less code, but trying to change the behavior of these functions is not possible as the code is not accessible to developers.
2. Using TeleUSE runtime libraries makes porting to other platforms impossible without having TeleUSE or a TeleUSE porting kit available.
3. Writing the dialog management part requires a process of learning a new 'D' scripting language.
4. Converters develop 'C' code with little or no documentation on the TeleUSE specific functions which makes maintenance and upkeep of the code very hard.

5. TeleUSE adds an additional effort for each new individual or group that deals with the code, because each developer must learn the tool.
6. Code that is developed using this process is in 'C' and cannot be changed to other languages.
7. TeleUSE provides limitations on which platforms and what windowing system it can run.
8. Use of TeleUSE implies additional costs and licensing requirements on this project.
9. Being a non-standard development environment and using propriety widget sets makes compatibility between TeleUSE generated code and other code an issue.

4.3 Next Phase in Development

Essentially, developers who wish to change the functionality of the application must know two things: the set of rules and interface operations that apply to the application, and the syntax and semantics of the implementation language and any other implementation tools. The next phase of the project involves looking into the functional or operational changes shown in section 4.1, and accommodating them in the design and development of a newer version of EIS. In Prototype 1 EIS subsystems other than the GUI are more or less ignored. Thus concerns related to integrating other subsystems with the GUI have not been taken into consideration. Incorporating these other subsystems of the EIS will cause some GUI specification changes, as the designer looks at details of

managing a local state sufficient to allow correct display and interpretation of specific EIS operations. That is, once the GUI determines what operation is invoked, it must forward an operation request to the another subsystem, like the Object Management Engine, for actual interpretation. For example, a class hierarchy may be represented in different forms or states for different subsystems. The GUI needs a state optimized for display, the OME uses a state to support object manipulation operations, and the ODBM uses a form for storage and retrieval operations. The first prototype ignores operational criteria for all the subsystems, looking only at broad functional goals. Consequently the first GUI subsystem cannot be easily integrated with other subsystems. The emphasis of prototype 1 is on the basic tools that is used to develop the GUI, and it's structure. Its objective of providing with a "shell" of windows and buttons gives users a chance to visualize the user interface. Because the new GUI would require changes in both window appearance and window dynamics, extensive change to the TeleUSE base and Visual Programming is required.

However after evaluation of the tool presented in section 4.2, a need to look at other options for continued development of the system becomes quite obvious. This software development process has now come to a point where a decision must be made whether to continue to use the GUI generator or change to some other development methods. Once the decision is made, some sort of planning must follow to determine what additional development or training steps need to be taken. This section looks at two options using some predefined factors and its impact on developers. One option is to use

TeleUSE; and the other is to switch to the use of standard X Window Programming techniques.

The basic approach using an interface builder like TeleUSE is that it assists developers in organizing the information, selecting appropriate interface object classes and their attributes, and placing selected interface objects in a dialog box or a menu in a meaningful, logical, and consistent manner. In contrast, the X Windows programming environment provides an elegant and extensive set of low level graphics and windowing utilities, along with higher level X-based libraries. Thus you can program in X via low-level Xlib calls, the X Intrinsic toolkit, and high-level widget display objects as provided by the Athena, Motif and Andrew toolkits. Most X applications are written in one of the higher level widget-based toolkits, as writing applications in Xlib is tedious. Following are some of the factors that are used to evaluate the use of TeleUSE vs. X.

Factor 1 - Cost analysis: This defines the cost of learning a development tool and producing the requisite software product. Developing a GUI application requires developers to learn multiple languages. A layout language is used to program the layout of the interface. The dialog control is programmed in a dialog language (typically an event language). The semantics of the application are programmed in a general purpose programming language (e.g. C or C++). In both options, the bulk of the application code is written using a general purpose programming language. Also, both these options adhere to an object-oriented paradigm to deal with the complexity of user interface software. Once the learning process is done, the next step involves coding time and

coding costs. Coding costs includes initial development costs, extensibility costs, maintenance costs, etc. There is also a cost for licensing the tool itself.

In TeleUSE, the layout is built using visual programming and the dialog is implemented using a dialog scripting language called 'D' scripts. This tool can also generate the user interface as Motif UIL code or as C code. Thus, TeleUSE requires learning the tool itself, the concept of visual programming and how it generates interface code, its specific dialog scripting language, knowledge of X, UIL and C or C++. TeleUSE requires less initial coding time, because of the visual programming mechanism that helps in developing the layout component of the user interface and the dialog management that requires very few lines of code. However, TeleUSE may require more learning time because it has more tools the developer must master. Though TeleUSE has less initial development costs, lack of good extensibility and difficult maintenance of generated code increases the overall coding time and costs.

In X window programming, arranging of widgets for the layout component and the dialog management requires writing code in X, user interface language (UIL) and/or programming in C or C++ using high level widget objects. The dialog management uses callbacks via eventhandlers to interface with the application code. Thus programming in X requires learning each of these languages. Initial coding takes longer because the presentation component and the dialog management is all done by writing code, as opposed to generating code in TeleUSE. However, because most of the code is written by

the developer and requires no mastering of special tools, overall coding time and costs for extensibility and maintenance are low.

Factor 2 - Maintenance of the code: This defines the ease with which the software can be extended or contracted to satisfy new requirements or can be corrected when errors or deficiencies are detected. Thus, for the code to be easy to maintain, it should be easy to understand, and modify.

TeleUSE generates code from the Visual Programming and the D scripts, adding considerable complexity. It also includes TeleUSE specific libraries that are not accessible. While this may make the code easy to generate, it also makes understanding and modifying the code hard. Coding correctness is based on how well the visual component and the dialog management code is converted using the interface builder provided by TeleUSE. The dialog management requires knowledge of a specific scripting language and its correctness is based on how well the developer understands the special scripting language.

X Window programming on the other hand makes the difficulty of maintenance roughly the same as original development, as the structure and libraries used in the code are relatively standard across applications. For X, the correctness is completely based on the capabilities and experience of the developer in using standard libraries, languages and the user interface software.

Factor 3 - Exploratory and incremental programming: Since there is a lack of a good set of rules to develop a complete interface from design in one cycle of

development, several iterations of prototypes and refinements are necessary. This factor defines the feasibility of this kind of programming. Exploratory and incremental programming are very important in interface development. Given the need for iteration, it would be desirable to have an interpretive language so that programs can be written and tested without experiencing compiler delays.

TeleUSE provides this concept of interpreter. Once the final interface is developed, it also allows the programs to be compiled. At compile time, TeleUSE requires X, Motif libraries, toolkit libraries, TeleUSE specific libraries and other system libraries.

Programming in X requires compiling of the programs to iterate through the development process. At compile time, X Window Programming requires X, Motif, toolkit libraries and other system libraries.

Factor 4 - Limitations of the developed product: This characterizes the deficiencies in the final product in terms of capabilities, compatibilites, system requirements, etc.

Being a non-standard GUI development environment, TeleUSE requires a specific porting kit to port to other platforms. Also, with so many UIMS (User Interface Management System) tools in the market, the chance that the TeleUSE vendor continues to develop and maintain future versions of the tool is a risk that a developer has to weigh against a standard development environment like X Window programming. Finally, the main shortcoming of interface builders like TeleUSE is their inability to specify

interfaces for objects that change at run time. Interface builders are excellent for constructing dialog boxes containing menus, buttons, sliders and other such widgets, provided that all the widget instances to be displayed are known at design time. For instance, the number of radio buttons in a group and their labels must be known at design time so the developer can arrange them on the screen as they are going to look to the end user. However if the set of choices is determined at run time, it is impossible at design time to specify the labels and the position of the buttons in a WYSIWYG editor like TeleUSE. The impact, in case there is a need for run time interface objects is to refine or edit generated code produced by interface builder to include this capability.

In contrast a GUI built with X window programming is relatively standard software that allows the application to be portable across a variety of UNIX platforms. There is no limitation on the ability to specify interfaces for objects that change at run time.

Factor 5 - Configurability of the UI part: This defines the capability to configure the GUI. Very often the end-users want to customize their screens to have a different look. These changes can be as simple as the color used for their screens or different menus, labels, or extend to individual differences related to cultural, ethnic, racial or linguistic backgrounds. Some of the examples include creating dialogs that use names and titles (Mr., Ms, Sir, Dr., etc.), different date and time formats, etiquette, tone and formality for interactions, etc. The concept of configurability clashes with the concept of consistency for user interface software. However, because the purpose of the software is

to attract novice, knowledgeable, and expert users, there needs to be a healthy balance of both concepts. TeleUSE lacks a way to easily embed this concept of configuration of the user interface. On the other hand, X window programming allows this capability.

After understanding these factors associated with each option, a decision can be made on the prototyping tool that is used to develop the next GUI. Development in X has less long term learning costs when a sequence of implementors is considered. For example, both development methods require knowledge of X. TeleUSE use requires that each developer understand the tool and learn a new scripting language, which adds to the learning process. Using X, the software developed has lower coding time, and also lower costs for future maintenance and extensibility. This is because the programming languages and libraries used are standard in the software industry. There is a much better chance of getting already trained and experienced developers for present and future iterations of development. The TeleUSE objective is rapid prototyping through visual programming, concise dialog scripting language and incremental programming using interpretive language. Since in TeleUSE the interface code is generated from layout language definitions provided by the developer and a similar generation of code is done from dialog scripting language, the code is possibly better optimized. But the impact is minimal for present computers with better processor speeds and cheap memory. So with an emphasis on developing the GUI with the least overall learning and coding costs, better maintenance, standard development environment, and long term portability, using the X window programming becomes an obvious choice.

CHAPTER 5

DISCUSSION

UI development is a high-risk business. To reduce risk, proper planning and development should be discussed. This helps to develop expertise, pointing out errors and limitations early. The developer can then concentrate on better approaches to development of this product and avoid making preliminary arbitrary decisions. This chapter discusses how the decision to switch to X impacts planning of the next spiral of development, in terms of various factors that are associated with them thereby entering quadrant 1 of a new spiral. At the end, this chapter emphasizes the importance of studying a software development life-cycle.

5.1 Quadrant 1 for this GUI development

Section 4.3 reviews a list of factors and compares the two options (TeleUSE vs. X) with respect to the factors. This section concentrates on systematically using these factors to plan on what needs to be done for each factor to get ready to proceed. Following are the impacts on planning for switching to X.

Factor 1 - Cost analysis: Switching from TeleUSE to X/Motif programming requires use of X, C, and/or UIL. The cost of learning these should be minimal, as all are standard and known, and required for effective use of TeleUSE anyway. Before recoding, the system should be first divided into conceptually understandable modules. The modules should be arranged in a hierarchical order dictating how modules can be invoked and how they interact with one another. Second, the developer should investigate the

common library routines and existing production modules that perform functions similar to those required. The impact is that initial recoding costs are high, because the next prototype development needs to be done from scratch, and parts such as the presentation component and the dialog management code may have to be completely redesigned.

Factor 2 - Maintenance of the Code: X uses a logical object-oriented paradigm to reduce the complexity of the user interface software. However to provide better maintainability and extensibility of the code, thorough planning and analysis of specifications need to be done to clearly separate the interface component, the dialog component and the application code. Also, the modules that make up these three components should allow the ability to change. Considering change possibilities helps the developer evaluate the degree of generality versus flexibility to be designed into the system. Generality allows a system to be used for a variety of changing functions without introducing modifications, while flexibility allows the system to be modified easily. An effective approach should include appropriate degree of both, and provide code that is relatively easy to maintain. Following are some of the general guidelines that should be used to guide the coding phase:

1. Use high-level programming languages wherever feasible.
2. Use only standard features of a programming language.
3. Limit the number of files each component accesses.
4. Document source code to explain the function that each module is to perform and choose descriptive data and procedure names.

5. Avoid hard-coded parameters.

6. Limit the size of the modules and write code that is readable, reliable and complies with coding standards.

Factor 3 - Exploratory and incremental programming: Component specifications are required to describe how software requirements are to be met. Specifications should include definitions for input/output formats, data structures, functional components and interfaces with other subsystems. The designer must use the specifications to identify subsets of functional and data structure components that are initially required and those that are required at later phase. The design and development should proceed in explicit steps. As little as possible should be decided at each step, and attempts should be made to make the easiest decisions first. For example, the designer should start with the presentation component of the GUI, make sure the layout of interface objects is done properly and according to the design, then proceed with the dialog management component and application code.

Factor 4 - Limitations of the developed product: Programming in X ensures a high degree of standardization of the user software developed. This includes common user interface features across multiple applications. It also helps users learn applications quickly. However, developing a product using X and Motif standards clashes with the look and feel of other popular interface standards like Microsoft Windows or Apple's interface. Arranging for portability becomes a challenge for developers if the system is

required to use different user interfaces standards across multiple software and hardware environments.

Factor 5 - Configurability of the UI part: Programming in X allows for the option of configurability of screens, fonts, colors, labels, menus, etc. provided proper design is done well ahead to accommodate user customizations.

After planning and evaluating the current EIS based on the factors above, development of the second version of EIS can be done using X-Window and the OSF/Motif widget set. This development is not described in detail here, though several key factors are mentioned. One of the major requirements is development of a tree widget class. Other key components are further development of other subsystems, like the OME, ORB etc. The complete new EIS system depends on these key aspects. This project reviews only the GUI components, with a thorough evaluation of both the operations and prototyping tools that have evolved over time.

5.2 Conclusions

It is important to note that, by focusing on the visual component and using a commercially available GUI builder, prototype 1 gives developers a way to rapidly get early user input on visual layouts of buttons and windows and assess the usability of the interface development tools. This development is achieved rapidly and with minimum labor costs. It translates a basic EIS framework into an implementation model that helps users see the proposed EIS system as a software tool. The developers better understand

the process of GUI prototyping along with the methods of achieving it. A complex subsystem like the GUI needs to start small to give the users and developers an easier way to conceptualize. It also provides a stepping stone for further iterative process of design and development. However, difficulties arise when a developer tries to reorganize the content or change the application interface object, especially when this process is cumbersome. Costs escalate, if a new development team is required to prototype, because prototyping process involves learning complex tools. Also a traditional problem facing UI development methodologies has been how to fit them in practical development life cycles. Bohem's spiral model, with it associated concept of risk emphasis, defines a pathway by which UI design concerns and methods can be incorporated into a larger system development methodology.

Taking it a step further, this approach gives a lot of relevance to everyday, real world software projects. We have a great deal to learn about producing and supporting well-engineered, useful software. Users are frustrated and antagonized by the introduction of software products that are difficult to use and that do not work as expected. Software engineers are at loss to understand why one project succeeds and the next one fails. Recording and studying software project case histories must become a required component of the software industry. We must recognize software engineering as an applied, and not a theoretical discipline. Defining software engineering principles and methodologies is only the first step. We must also evaluate their utility in practice. A

major problem blocking software development methods is the lack of reported software project case histories.

The methodology used in early EIS development process, especially for UI development is a good case study for using principles to work in practice in trying to improve the software development life cycle. Rapidly developing a prototype for an application program that is hard to conceptualize is a good first step. This enhances communication about the proposed application program as a prototype creates a common baseline or reference point from which potential problems and opportunities are identified. Users tend to be more enthusiastic with a project in which they are involved through the use and evaluation of prototypes. It seems that users are very good at criticizing an existing system (i.e., a prototype) but are not usually good at anticipating or articulating needs. The earliest version that users can experiment with, whether prototype or real product, can cause them to change their view about what they want the system to do. With this concept in mind, the best option a developer can use is to do prototyping at an early stage and not spend all the resources in developing a final product that may or may not be the one users want.

Boehm explains that fundamental to the process of producing quality software is a commitment on the part of the software developer to continually seek ways in which to improve the software product and its production. This project uses a software development process model, like the Bohem's spiral model with emphasis on evaluation as a valuable feedback, for several reasons. First, it gives the developer feedback on the

initial perception of software quality. This feedback will help reinforce the notion that the quality of software does not entirely lie in its form, but also in its utility, maintainability, etc. Second, it is useful to the developer in planning future development efforts. It could point out the shortcomings in this product or the tool used to develop this product. Third, it can be used as a learning tool to help developers better understand what it takes to get a better approach in development of a software product.

BIBLIOGRAPHY

Bohem, B. "A Spiral Model of Software Development and Enhancement", IEEE Computer, Vol. 21, No. 5 (May 1988), pp 61-71.

Booch, G. Object Oriented Design with Applications, The Benjamin/Cummings Publishing Company, Inc, 1991.

Deborah, H. and H. Rex Harston. Developing User Interfaces: Ensuring Usability Through Product and Process, John Wiley and Sons, 1993.

Ford, R., R. Righter, T. Duce, V. Hemige and D. Thompson. "EIS: A Network-Accessible Repository for Ecosystem Modelers and Managers," Proceedings of Decision Support - 2001, Resource Technology 1994 Symposium.

Gould, J., D. Boies, and C. Lewis. "Making usable, useful productivity-enhancing computer applications, Communications of the ACM, (January 1991), pp 75-85.

McClure, C. L. Managing Software Development and Maintenance, Van Nostrand Reinhold Company.

Myers, Brad A. Languages for Developing User Interfaces, Jones and Bartlett Publishers, 1992.

Pressman, R. S., Software Engineering: A Practitioner's Approach, McGraw-Hill Publishing Company, 1992.

Riemer, Y., R. Ford, and N. Wilde. "Merging Task-Centered UI Design With Complex System Development: A Risky Business.

Shneiderman, B. Designing the User Interface: Strategies for Effective Human-Computer Interaction, Addison Wesley Longman, Inc, 1998.

TeleUSE Reference Manuals, 1 & 2, TeleSoft, 1991.

UIST, Proceedings of the ACM SIGGRAPH: Symposium on User Interface Software and Technology, 1990.

Watson, M. Portable GUI Development with C++, McGraw-Hill, Inc, 1993.